Relatório sobre a implementação de um analisador léxico e um analisador sintático para a linguagem ZSharp (Z#).

1. Sobre este trabalho

Este trabalho foi implementado com o intuito de servir como um estudo sobre a disciplina de compiladores, ministrada pelo professor Alexandre Agustini na Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS).

Neste, foi realizado o desenvolvimento de um analisador léxico e um analisador sintático para um subconjunto reduzido da linguagem C#, denominado Z# (http://dotnet.jku.at/courses/CC/).

1.1. Autores

Italo Lobato Qualisoni - 12104861-5 - italo.qualisoni@acad.pucrs.br

William Henrihque Martins - 12104965-4 - william.henrihque@acad.pucrs.br

2. Ambiente de desenvolvimento

Para o desenvolvimento deste, foi utilizado o sistema operacional Linux, mais especificamente utilizando as distribuições Ubuntu e Lubuntu.

3. Estrutura de diretórios

Os arquivos e diretórios do projeto encontram-se organizados da seguinte forma:

```
./bin/                  - binários relacionados ao projeto
  JFlex.jar             - cópia local do programa JFlex
  yacc.linux            - cópia local do programa Byacc/J

./dist/                 - diretório que recebe o projeto inteiro compilado usando a compilação pelo Makefile

./doc/                  - documentação do projeto
  Relatorio.doc         - relatório da implementação da solução

./samples/              - diretório contendo programas exemplo

./src/                  - código fonte do projeto
  lexical/              - código fonte relacionado a parte léxica
      zsharp.flex       - especificação léxica para o analisador léxico

  syntactic/            - código fonte relacionado a parte sintática
      ZSharpASAR.y      - especificação sintática para o analisador sintático

./tests/                - diretório contendo testes da solução
  test_all.sh           - script para testar todos os arquivos contidos na pasta samples

./Makefile              - arquivo makefile para a compilação do projeto
```

4. Instruções de compilação

4.1. Softwares utilizados

Utilizamos basicamente dois programas para a geração dos analisadores:

Para o léxico, foi utilizado o JFlex.

Para o sintático, foi utilizado o Byacc/J.

4.2. Realizando a compilação do projeto

Para uma compilação simples, basta utilizar o comando `make` na raiz do projeto, tal comando fará com que o projeto seja compilado para a pasta 'dist'.

4.3. Problemas de compilação utilizando Linux 64 bits

Encontramos um pequeno problema de compilação utilizando `make` no Ubuntu 13.04 64 bits. O problema se dá devido ao fato de que o programa yacc.linux é um programa de 32 bits. Para resolver, basta instalar o pacote `ia32-libs-multiarch` para resolver o problema. Para isso, simplesmente execute o comando a seguir:

```
sudo apt-get install ia32-libs-multiarch
```

5. Utilizando o analisador léxico

Basta utilizar o comando `java Yylex <nome-do-arquivo>` para que o analisador léxico seja executado. A saída desse programa é no console, portanto, para jogar a saída para um arquivo basta utilizar o comando `java Yylex <nome-do-arquivo> >> <meu-arquivo-de-saida>`.

Ainda existe uma outra maneira utilizando o modo interativo. Para tal, basta executar o comando `java Yylex` que será aberto o terminal interativo. Então, basta digitar a entrada e apertar a tecla <ENTER>, dessa forma, cada token reconhecido será mostrado da seguinte forma:

token: NUMERO_DO_TOKEN <TOKEN>

Onde "NUMERO_DO_TOKEN" indica o número do token reconhecido e "TOKEN" indica o que foi digitado.

6. Utilizando o analisador sintático

Basta utilizar o comando `java Parser <nome-do-arquivo>` para que o analisador sintático seja executado. A saída desse programa é no console, portanto, para jogar a saída para um arquivo basta utilizar o comando `java Parser <nome-do-arquivo> >> <meu-arquivo-de-saida>`.

Ainda existe uma outra maneira utilizando o modo interativo. Para tal, basta executar o comando `java Parser` que será aberto o terminal interativo. Então, basta digitar a entrada e apertar a tecla <ENTER>.

7. Testes da solução

Existem 10 exemplos de código Z# que foram fornecidos para download no site http://dotnet.jku.at/courses/CC/ e estão localizados localmente na pasta 'samples'. Dentre os exemplos, apenas 1 possui erros no código, portanto, é o único arquivo que não irá compilar, apresentando erros de compilação.

Os arquivos da pasta samples encontram-se da seguinte forma:

```
samples/
  AllProdsWithErrors.zs         - Arquivo com erros de código. NÃO deve compilar.
  AllProds.zs                   - Arquivo correto. Deve compilar.
  BubbleSort.zs                 - Arquivo correto. Deve compilar.
  P.zs                          - Arquivo correto. Deve compilar.
  QuickSort2.zs                 - Arquivo correto. Deve compilar.
  QuickSort.zs                  - Arquivo correto. Deve compilar.
  StudentList.zs                - Arquivo correto. Deve compilar.
  TestBreak.zs                  - Arquivo correto. Deve compilar.
  TestBuiltInFuncs.zs           - Arquivo correto. Deve compilar.
  TestNoRetFunc.zs              - Arquivo correto. Deve compilar.
```

7.1. Utilizando testes automatizados

Dentro da pasta 'tests', fizemos a criação de um script em bash para o teste automático de todos os exemplos localizados na pasta 'samples', tal script foi chamado de 'test_all.sh'. Dessa forma, esse script faz a tentativa de compilação de todos os arquivos da pasta 'samples', mostrando a mensagem 'done!' quando a compilação é finalizada e, em caso de erro, mostrando a saída de erro que o compilador gera.

Para executar esses testes, basta entrar na pasta 'tests' e executar o script 'test_all.sh'.

7.2. Execução do teste automatizado

Rodando o comando `./test_all.sh`, obtivemos a seguinte saída:

```
=============== Executing tests ===============

Testing [AllProdsWithErrors.zs] - this file must NOT compile
Error: syntax error
Line: 3
Column: 17
Error: stack underflow. aborting...
Line: 3
Column: 17
done!
Testing [AllProds.zs] - this file must compile
done!
Testing [BubbleSort.zs] - this file must compile
done!
Testing [P.zs] - this file must compile
done!
Testing [QuickSort2.zs] - this file must compile
done!
Testing [QuickSort.zs] - this file must compile
done!
Testing [StudentList.zs] - this file must compile
done!
Testing [TestBreak.zs] - this file must compile
done!
Testing [TestBuiltInFuncs.zs] - this file must compile
done!
Testing [TestNoRetFunc.zs] - this file must compile
done!
```

Com esses resultados, mostramos que a nossa solução funciona para esses 10 casos, mostrando assim que a implementação está correta e funciona conforme especificado.

7.3. Execução do teste automatizado a partir do comando `make test`

O comando `make test` irá recompilar a estrutura lexica e sintatica e executar em seguida o comando `./test_all.sh`, gerando a saída da recompilação seguida dos resultados dos testes presentes na pasta 'samples'.

8. Conclusões

Este trabalho nos mostrou como realmente funcionam duas partes muito importantes nos compiladores, que são a análise léxica e a análise sintática. Nos mostrou ainda que essas duas partes andam muito juntas e são dependentes uma da outra, dessa forma, quando uma delas possui algum tipo de erro, é bem provável que a outra, por mais que pareça estar certa, não vai funcionar corretamente.